University of Petra

307307
Generative AI

Introduction to LLMs, Transformers and Context Aware Embeddings

What Are Large Language Models (LLMs)?

- LLMs are neural networks trained on large text corpora to understand and generate human language.
- They learn statistical patterns, grammar, semantics, and world knowledge directly from data.
- Built on the Transformer architecture, which relies on self-attention to capture relationships between all words in a sequence.
- Capable of performing many tasks without task-specific training, such as answering questions, writing code, summarizing text, and translating languages.
- Examples: GPT, PaLM, LLaMA, Claude

How LLMs Are Trained

- **Pretraining:**
  - Models see massive amounts of text and learn by predicting masked or next tokens.
  - This builds general language understanding and knowledge of syntax, semantics, and facts.
- **Fine-tuning:**
  - Models can be adjusted for specific tasks (classification, summarization, translation).
- **Instruction Tuning and RLHF:**
  - Models learn to follow human instructions using curated datasets and human feedback.
  - Ensures outputs are helpful, safe, and aligned with user intent.
- The scale of data and parameters allows the emergence of advanced reasoning and generative abilities.

Pretraining vs. Fine-Tuning

Pretraining Example: A base LLM like GPT is trained on **trillions of tokens** from books, websites, articles, and code.
It learns:

- How sentences are structured
- How logic works
- Broad world knowledge
- General reasoning skills
  This is like teaching the model *all of human language*.

Fine-Tuning Example: A company wants a **customer-support chatbot** for their bank.
They take the pretrained model and fine-tune it using:

- 5,000 chat transcripts
- Bank-specific FAQs
- Compliance rules and preferred tone
  After fine-tuning, the model learns:
- Bank terminology
- Policies for loans, fees, accounts
- Approved response style
- How to avoid giving unauthorized financial advice
- This process takes hours instead of months and costs thousands instead of millions.

How LLMs Work at Inference Time

- Input text is tokenized and converted into embeddings.
- The model processes the input through multiple Transformer layers to build a contextual representation.
- For generation, the model predicts one token at a time, using masked self-attention to avoid “seeing the future.”
- Decoding strategies (greedy, beam search, sampling, top-k, top-p) shape the creativity and quality of the output.
- The model continues generating tokens until an end-of-sequence token or stopping condition is reached.

What LLMs Can Do and Their Limitations

- **Strengths:**
  - Text generation, explanation, summarization, reasoning, coding, translation, question answering.
  - Adaptable to many tasks via prompting, not just training.
- **Limitations:**
  - Can produce incorrect or fabricated information (hallucination).
  - Lack true understanding or grounded world perception.
  - Sensitive to prompt phrasing and context.
  - Require substantial computing resources for training and inference.
- LLMs are powerful tools, but outputs must be interpreted with domain knowledge and verification.

Key Generation Settings in Language Models

Context Window

- The maximum amount of text the model can consider at once.
- A larger context window lets the model remember more of the conversation or document, improving coherence over long inputs.
- Once the limit is reached, older text is no longer considered.

Temperature

- Controls randomness in generation.
- Low temperature (e.g., 0.2) makes the model more focused and deterministic.
- High temperature (e.g., 1.0) makes the model more creative and varied.

Top-N / Top-K Sampling

- Limits the model to choosing from the top K most likely next tokens.
- Lower K makes output more predictable.
- Higher K adds more variety while still avoiding extremely unlikely tokens.
- If you want, I can format this as bullet points for a real slide deck or export it to PowerPoint.

Introduction to the Transformer Architecture

Transformers – Architecture and Principles

What is a Transformer?

A neural network architecture based on self-attention mechanism. It processes all tokens in parallel and models long-range dependencies efficiently.

Introduced in "Attention Is All You Need" (Vaswani et al., 2017), it became the foundation for most modern language models.

The transformer model consists of two main parts:

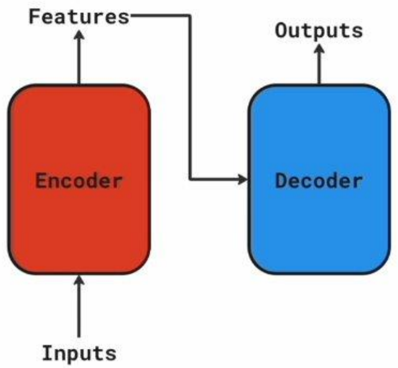
1. Encoder:

- The encoder processes the input sequence and generates an encoded representation of it.
- This representation captures the contextual information of the input tokens.

2. Decoder:

- The decoder takes the encoder's output and generates the final output sequence.
- It does this by predicting one token at a time, using the encoded representations and previously generated tokens.

Both the encoder and decoder are composed of multiple identical layers—typically six layers in the original transformer architecture—allowing for deep learning and complex feature extraction.

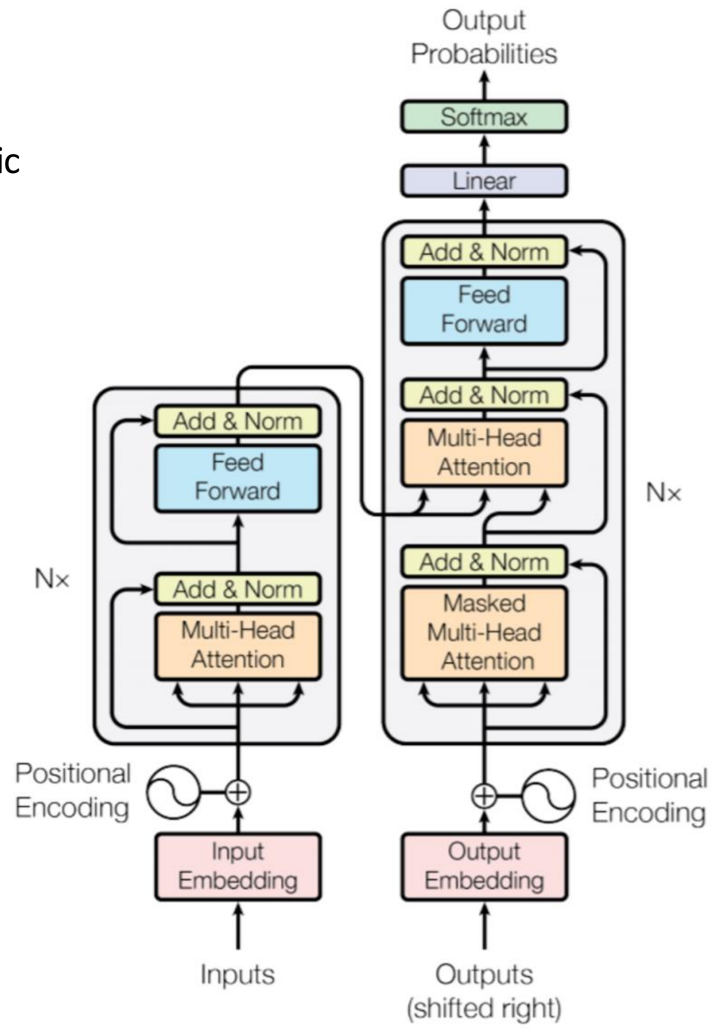


"Hello world" -> encoder -> knowledge -> -> decoder -> "Hola mundo"

Transformers – Architecture and Principles

Transformer Core Components

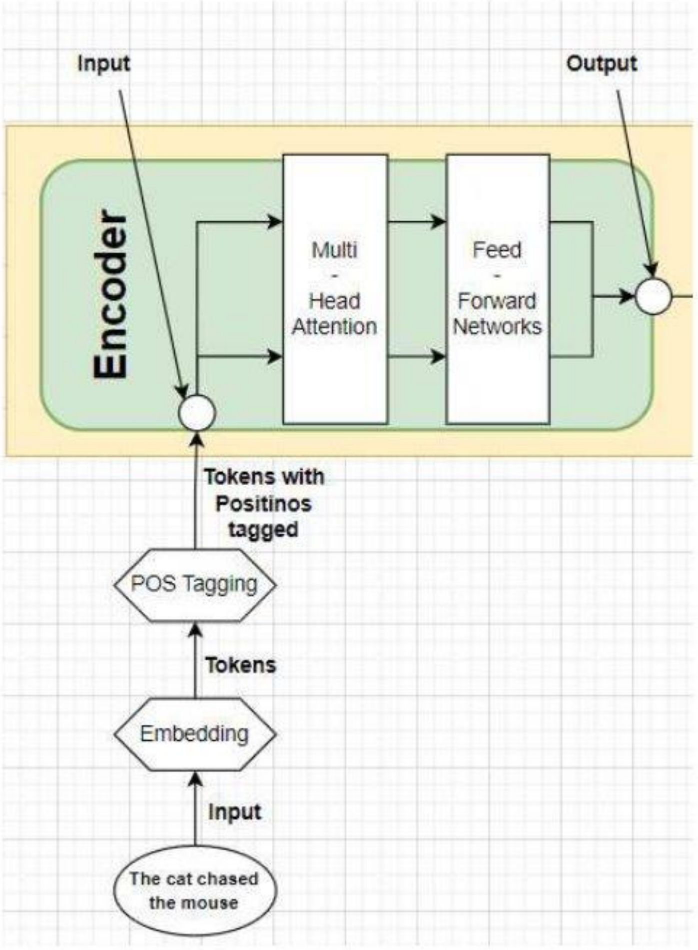
- **Embeddings:** Map input tokens into dense vector representations that encode semantic and syntactic information.
- **Positional Encoding:** Injects information about token order using sinusoidal or learned patterns, enabling the model to understand sequence structure.
- **Multi-Head Self-Attention:** Lets each token attend to every other token, with multiple heads capturing different types of relationships (e.g., subject–verb, entity–modifier).
- **Feed-Forward Networks:** Apply two linear transformations with a nonlinearity at every position, enabling deeper feature transformation beyond attention.
- **Layer Normalization:** Normalizes intermediate activations to stabilize training and improve convergence.
- **Residual Connections:** Add the input of each sublayer to its output, preserving information and enabling effective gradient flow in deep stacks.



A Summary of how the Transformer Works

The Encoding Process:

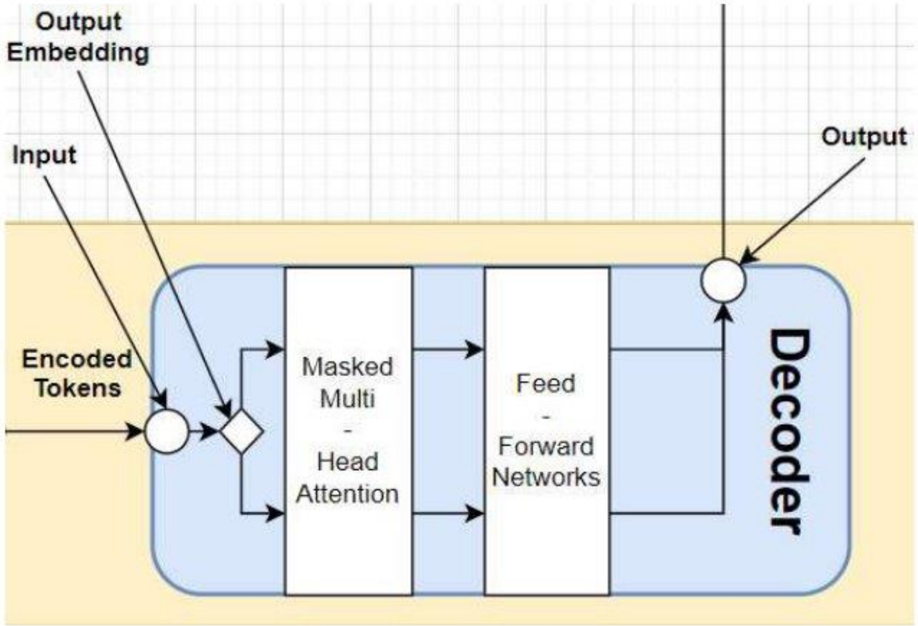
1. The input sentence “The cat chased the mouse” is first tokenized into individual tokens such as “The”, “cat”, “chased”, “the”, “mouse”.
2. Each token is converted into a vector embedding, a learned numerical representation that captures semantic information about the word.
3. A positional encoding is added to each embedding to provide the model with information about token order, which the self-attention mechanism alone does not capture.
4. The resulting vectors are passed through a stack of encoder layers, each containing multi-head self-attention and a position-wise feed-forward network.
5. In each layer, self-attention allows every token to attend to all other tokens in the sentence, enabling the model to capture dependencies such as the relationship between “cat”, “chased”, and “mouse”.
6. As the representations flow through successive layers, the encoder builds increasingly rich contextual embeddings that summarize the meaning and structure of the entire input sentence.



A Summary of how the Transformer Works

The Decoding Process:

1. The decoder receives two inputs: the encoder's output (a contextual representation of the source sentence) and the previously generated target tokens, which are shifted right and begin with a special start-of-sequence token (e.g., `<bos>` or `<start>`).
2. In the masked multi-head self-attention layer, each target position attends only to earlier target tokens; the causal mask prevents access to future positions and ensures autoregressive prediction during training and inference.
3. In the encoder–decoder (cross) attention layer, the decoder queries the encoder's output, allowing each target token to focus on the most relevant parts of the source sentence for accurate translation.
4. A position-wise feed-forward network further transforms each position's representation, adding nonlinearity and helping the model capture complex relationships.
5. The decoder then projects the processed representations through a linear layer and softmax to produce a probability distribution over the vocabulary, selecting the next token; this token is appended to the sequence and becomes part of the "previously generated tokens" for the next decoding step.
6. Repeating this process yields the target sentence token by token, such as "Le", "chat", "a", "poursuivi", "la", "souris", forming the translation "Le chat a poursuivi la souris



Introduction to the Attention Mechanism

- Attention allows each token in a sequence to decide which other tokens are most relevant for interpreting its meaning or generating the next representation
- The mechanism works by comparing a token's query vector to the key vectors of all other tokens, producing similarity scores that reflect how strongly each pair is related
- These scores are normalized with softmax, creating a probability distribution that determines how much each value vector contributes to the token's updated representation
- Multi-head attention repeats this process in parallel, enabling the model to capture different linguistic patterns such as grammatical structure, long-range dependencies, and contextual cues
- Visualizations such as heatmaps reveal how an individual head allocates attention and provide insight into which relationships the model has learned during training

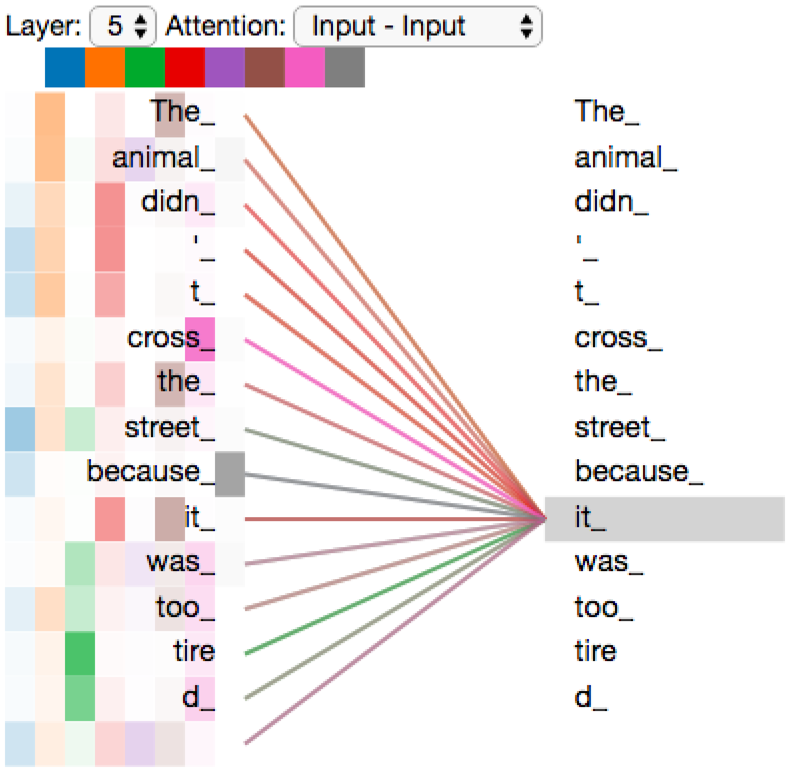
The Attention Mechanism

- The heatmap displays attention weights, where each row shows how one token distributes its focus across all other tokens
- Brighter cells indicate higher attention values, revealing which tokens are considered most relevant for understanding or generating the current token
- Rows sum to one, meaning each token assigns a probability distribution over all other tokens based on learned query–key similarities
- The visualization captures how a single attention head processes structure and meaning at this point in training
- The head shows sharply peaked syntactic attention, such as "the → cat = 0.90", "cat → chased = 1.00", and "chased → the = 0.98", indicating it has learned strong determiner–noun and noun–verb pairings
- Pronouns and auxiliaries distribute attention more broadly, as seen with "it → the = 0.23", "it → was = 0.17", and "it → chased = 0.12", showing the head gathers context across the sentence rather than relying on a single token
- Special tokens behave as expected: "[CLS] → [CLS] = 0.41" and "[CLS] → [SEP] = 0.36" reflect global-aggregation behavior, while "[PAD]" spreads very small values like "[PAD] → the = 0.15" and "[PAD] → cat = 0.33", showing padding is largely ignored
- Taken together, these patterns indicate the head mixes strong local grammatical dependencies with softer semantic context gathering, a typical role for mid-layer attention that supports coherent sentence understanding

	[CLS]	the	cat	chased	the	mouse	[SEP]	it	was	a	fast	chase	[SEP]	[PAD]
[CLS]	0.41	0	0.05	0	0.03	0.06	0.02	0.06	0.15	0.04	0.07	0.02	0.09	0
the	0	0	0	0.01	0.9	0	0.05	0.03	0	0	0	0	0	0
cat	0	0	0	1	0	0	0	0	0	0	0	0	0	0
chased	0	0	0	0	0.98	0	0.01	0.01	0	0	0	0	0	0
the	0.97	0	0	0	0	0	0	0	0.02	0	0	0	0	0
mouse	0	0.04	0	0.78	0.06	0	0.08	0.01	0	0	0	0.01	0	0
[SEP]	0.04	0.08	0.1	0.11	0.03	0.08	0.11	0.06	0.06	0.07	0.05	0.16	0.06	0
it	0.23	0	0.12	0	0	0.17	0.02	0.03	0.14	0.12	0.05	0.06	0.06	0
was	0.13	0.03	0.09	0.02	0.05	0.09	0.08	0.09	0.11	0.07	0.08	0.07	0.09	0
a	0	0.04	0	0.51	0.26	0	0.14	0.02	0	0	0.01	0.01	0.01	0
fast	0	0.09	0	0.67	0.19	0	0.02	0.01	0	0	0.01	0.01	0	0
chase	0.54	0	0	0	0.23	0	0	0.07	0.07	0	0.04	0	0.04	0
[SEP]	0.36	0	0.07	0	0.01	0.1	0.02	0.05	0.16	0.07	0.06	0.02	0.08	0
[PAD]	0.02	0.15	0.03	0.33	0.18	0.02	0.07	0.05	0.02	0.02	0.04	0.04	0.03	0

Interpreting Token-to-Token Attention Visualizations

- This diagram shows which input tokens a specific token is attending to at a given layer and head
- Each colored line represents the strength of attention from the selected token (e.g., “it_”) to another token in the sequence
- Stronger, more saturated lines indicate higher attention weights, revealing where the model is focusing its contextual understanding
- Different color bands typically represent different attention heads, each capturing a distinct linguistic pattern
- This view helps diagnose how contextual references, pronouns, dependencies, and sentence structure are being tracked by the model
- The selected token “it_” attends heavily to multipleearlier tokens such as “The_”, “animal_”, “didn_”, and“cross_”, indicating the model is gathering broadcontext
- The spread of attention across the sentence impliesthat the model is not guessing based on nearby wordsalone, but forming a coherent interpretation of theentire event being described

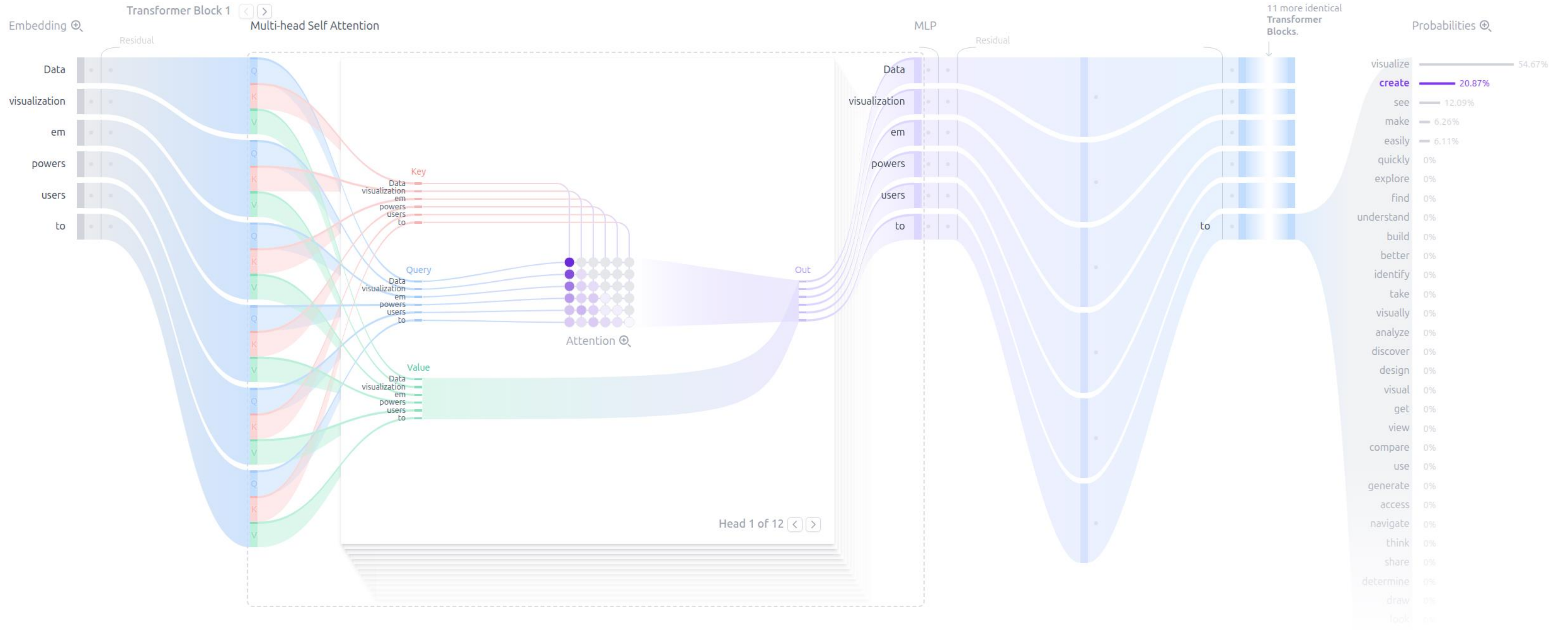


Number of Parameters - Original Transformer Model

Component	Parameter	Formula / Size	Total Parameters
Input	Token embedding	Vocab_Size × d_model = 37000 × 512	≈ 18.94M
	Positional encoding (fixed)	n × d_model	Not learned (original paper used fixed)
Attention (per layer)	Q/K/V weights per head	3 × d_model × d_k = 3 × 512 × 64	98,304
	Output projection	d_model × d_model = 512 × 512	262,144
	Total per Multi-Head block	–	≈ 360K
Feed-Forward (per layer)	Linear 1: 512 × 2048	–	1,048,576
	Linear 2: 2048 × 512	–	1,048,576
	Total FFN per layer	–	≈ 2.10M
LayerNorm	2 × γ, β per layer	2 × d_model = 2 × 512	1,024
Encoder Block Total	–	Attention + FFN + LayerNorm	≈ 2.46M
Encoder Total (6 layers)	–	6 × 2.46M	≈ 14.76M
Decoder Total (6 layers)	Similar structure + cross attention	≈ 2.6M per layer	≈ 15.6M
Output Layer	d_model × Vocab_Size = 512 × 37000	tied/shared with embedding	≈ 18.94M
Total Model Parameters	–	Encoder + Decoder + Embedding + Output	**≈ 65M**

Transformer Explainer

https://poloclub.github.io/transformer-explainer/



Context Aware Embeddings

Contextual Word Embeddings (BERT and GPT)

Key Innovation:

Unlike static embeddings Word2Vec, contextual models generate **different vectors for the same word** depending on its context.

Approach

- Uses deep, pre-trained neural networks (often transformer-based)
- Embeddings are derived from entire sentences, capturing syntax and semantics dynamically

Examples

- **BERT - Bidirectional Encoder Representations from Transformers (2018)**: Transformer-based neural networks trained with masked language modeling and next sentence prediction
- **GPT - Generative Pre-training Transformer (2018):** Transformer-based unidirectional language model focused on generation.

Characteristics

- Embeddings are **context-sensitive** (e.g., “bank” in “river bank” vs. “savings bank”)
- Each word is embedded based on its role in the sentence.
- Embeddings vary for the same word depending on its position and meaning.
- Significantly improve performance on downstream NLP tasks.

BERT: Bidirectional Encoder Representations from Transformers

- Developed by Google in 2018
- A **pre-trained language model** based on the **Transformer encoder**
- Reads text **bidirectionally**, enabling deep contextual understanding

Key Ideas

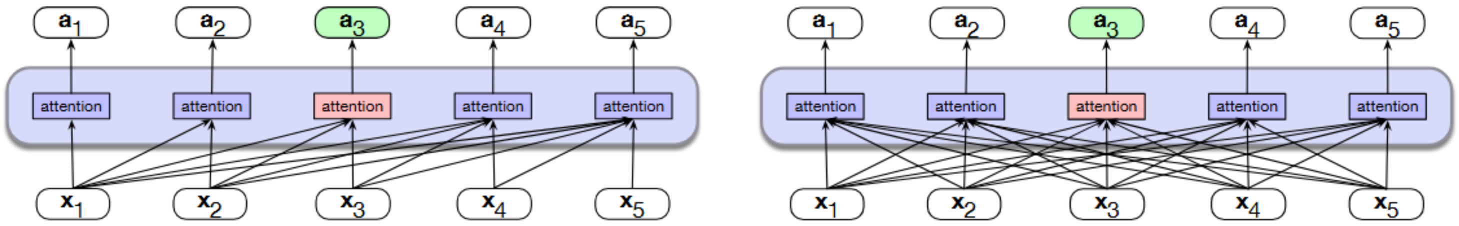
- Uses only the **encoder** stack of the Transformer
- Pre-trained on large text corpora, then fine-tuned on specific tasks

Pretraining Objectives

- **Masked Language Modeling (MLM)**: Predict randomly masked words in a sentence
- **Next Sentence Prediction (NSP)**: Predict if one sentence follows another

Applications

- Sentiment Analysis
- Question Answering
- Named Entity Recognition
- Text Classification
- Semantic Search



a) A causal self-attention layer b) A bidirectional self-attention layer

More About BERT

Key Advantages

- Contextual embeddings: Word meanings change based on context
- Captures long-range dependencies
- Pre-trained on massive datasets → Transfer learning
- State-of-the-art performance on 11 NLP tasks when released

How BERT Works:

- **Multiple stacked encoder layers** with self-attention mechanisms
- **No decoders** - purely focused on understanding input
- **Captures relationships** between words and their context
- **Powerful context learning** - understands word relationships in sentences
- **Meaningful representations** - transforms text into useful numbers

Architecture Overview

```
Input: [CLS] The cat sat on the mat [SEP]
              ↓
    Token Embeddings
    Position Embeddings
    Segment Embeddings
              ↓
    Transformer Encoder (12 or 24 layers)
              ↓
    Contextual Representations
```

Key Components

- **[CLS]**: Classification token (sentence-level tasks)
- **[SEP]**: Separator token (between sentences)
- **Multi-head attention**: Allows model to focus on different positions
- **Feed-forward networks**: Process attention outputs

BERT's Training Data

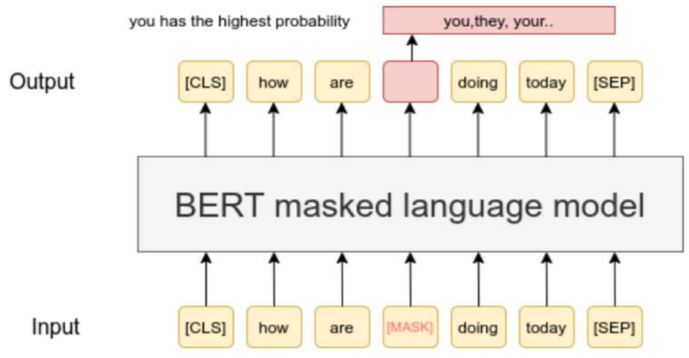
Pre-training Foundation

- **Training corpus**: 3+ billion words
  - English Wikipedia
  - ~10,000 unpublished books
- **Clean, large-scale dataset** for comprehensive language learning
- **Pre-training approach** to learn language patterns and context

Training Objective 1 - Masked Language Modeling

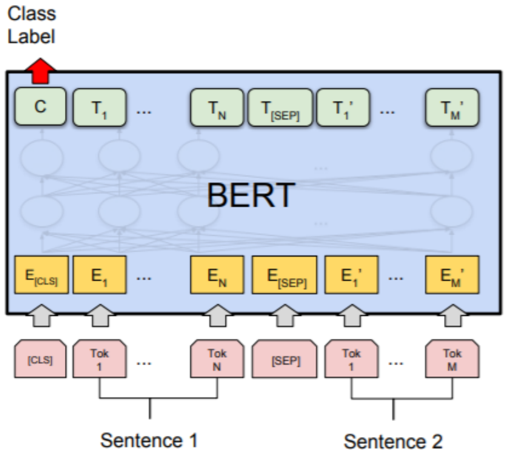
Learning Through "Fill in the Blanks"

- **Random word masking**: Some words are hidden during training
- **Context-based prediction**: BERT predicts masked words using surrounding context
- **Example**: "I love eating _____ in my fruit salad" → "lemons"
- **Key benefit**: Learns word relationships and contextual understanding
- **Powerful concept**: Used in many other AI applications



Training Objective 2 - Next Sentence Prediction

- **Title: Understanding Sentence Relationships**
- **Sentence pair analysis**: Given two sentences, determine if B follows A
- **Logical connection assessment**: Analyzes context and content
- **Example**:
  - A: "The cat climbed the tree"
  - B: "It was trying to catch a bird" ✓ (follows logically)
  - C: "The weather is nice today" ✗ (unrelated)
- **Note**: Later removed in newer models (MLM proved sufficient)



(a) Sentence Pair Classification Tasks: MNLI, QQP, QNLI, STS-B, MRPC, RTE, SWAG

BERT's Ideal Use Cases

Where BERT Excels

- **Text Classification**: Sentiment analysis, topic classification, spam detection
- **Named Entity Recognition**: Identifying people, organizations, locations, dates
- **Extractive Question Answering**: Finding answers within provided context
- **Semantic Similarity**: Measuring similarity between sentences/paragraphs
- **Key advantage**: Perfect for understanding tasks, not generation

Example: Print out BERT Embeddings

```
from transformers import BertTokenizer, BertModel
import torch

# Load pretrained BERT
tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')
model = BertModel.from_pretrained('bert-base-uncased’)

# Sentence
sentence = "He went to the bank to deposit money.“

# Tokenize
inputs = tokenizer(sentence, return_tensors='pt’)

# Get outputs
with torch.no_grad():  # No training, just inference
    outputs = model(**inputs)

# Get the hidden states (embeddings)
embeddings = outputs.last_hidden_state  # Shape: (batch_size, sequence_length,

# (hidden_size)
print(embeddings.shape)  # Example output: torch.Size([1, 11, 768])
```

Open in Colab

GPT – Overview and Architecture

What is Generative **P**re-trained **T**ransformer (GPT)?

- A family of **Transformer-based language models** developed by OpenAI
- Uses only the **decoder stack** of the original Transformer architecture
- Trained with **causal (autoregressive) language modeling** to predict the next token
- Focuses on generating human-like text (unlike BERT's understanding focus)

Training Objective

- Predict the next token in a sequence

GPT Variants

- **GPT-1 (2018, 120M Parameters)**: Introduced the pretrain-then-finetune paradigm
- **GPT-2 (2019, 1.5B Parameters)**: Scaled up model size, trained on web-scale data, Last publicly available weights
- **GPT-3 (2020, 175B Parameters)**: Enabled in-context learning, 100x boost, ~800GB file size
- **GPT-4 (2023, ~1T parameters (estimated))**: Multimodal, stronger reasoning and generalization

Applications (Wide range of NLP tasks through text generation)

- Text generation (e.g., chat, storytelling, code)
- Summarization
- Translation
- Question answering
- Semantic search and reasoning tasks

GPT's Architecture Deep Dive

How GPT Works

- **Multiple stacked decoder layers** with self-attention mechanisms
- **No encoders** - purely focused on text generation
- **Unidirectional processing** - considers only left-side context
- **Next word prediction** based on preceding words
- **High-quality text generation** from learned context patterns
- **Contextual understanding** combined with text creation capability

GPT Training Data

Pre-training Foundation

- **Massive, diverse datasets** including:
  - Web pages
  - Books and articles
  - Billions of words total
- **Different datasets** for each GPT version
- **Large-scale exposure** to language patterns and context
- **Quality varies** but emphasis on diversity and scale

Causal Language Modeling

GPT's Single Training Objective

- **Core task**: Predict the next word in a sequence
- **Method**: Uses context from preceding words only
- **Example**: "I love drinking fresh ___" → "lemonade"
- **Key benefits**:
  - Learns word relationships and context
  - Develops language patterns
  - Enables coherent text generation
- **Simplicity**: No masked language modeling or next sentence prediction

GPT's Ideal Use Cases

Where GPT Excels

- **Text Generation**: Story creation, creative writing, conversations
- **Instruction Following**: Responding to prompts and commands
- **Translation**: Converting text between languages
- **Summarization**: Creating concise summaries of longer texts
- **Code Generation**: Programming assistance and code completion
- **Versatile Applications**: Any task involving text output

GPT vs BERT Architecture

Aspect	GPT	BERT
Architecture	Decoder-only	Encoder-only
Text Processing	Unidirectional (left-to-right)	Bidirectional
Primary Goal	Text generation	Text understanding
Context	Previous words only	All surrounding words
Use Cases	Generation tasks	Classification/extraction

What is Hugging Face? 🤔

- **A company and a community platform** focused on democratizing Artificial Intelligence, especially Natural Language Processing (NLP) and Machine Learning (ML).
- Often called the "**GitHub for Machine Learning**."
- **Mission**: To make state-of-the-art ML models, datasets, and tools accessible to everyone.
- Started in 2016, initially with a chatbot app, then pivoted to open-source ML.

What does hugging face provide?

1. **Accessibility**: Provides easy access to thousands of pre-trained LLMs.
2. **Standardization**: Offers standardized tools and interfaces for working with different models.
3. **Collaboration**: Fosters a vibrant community for sharing models, datasets, and knowledge.
4. **Innovation**: Accelerates research and development in the LLM field.
5. **Ease of Use**: Simplifies complex ML workflows, from data preparation to model deployment.

Core Components of the Hugging Face Ecosystem

- **Hugging Face Hub**:
  - The central place to find, share, and collaborate on models, datasets, and ML applications (Spaces).
  - Over 1.7 million models, 75,000+ datasets!
- **Transformers Library**:
  - Python library providing thousands of pre-trained models for NLP, Computer Vision, Audio, and more.
  - Supports PyTorch, TensorFlow, and JAX.
  - Makes downloading, training, and using state-of-the-art models incredibly simple.
- **Datasets Library**:
  - Efficiently load and process large datasets.
  - Optimized for speed and memory, built on Apache Arrow.
  - Access to a vast collection of public datasets.
- **Tokenizers Library**:
  - Provides high-performance tokenizers crucial for preparing text data for LLMs.
  - Offers various tokenization algorithms and pre-trained tokenizers.

The Model Hub

Over 1,700,000+ Models Available

Popular Model Categories:

- **Text Generation**: GPT, LLaMA, Mistral, CodeLlama
- **Text Classification**: BERT, RoBERTa, DeBERTa
- **Question Answering**: BERT-based models
- **Translation**: T5, mT5, NLLB
- **Code Generation**: CodeT5, StarCoder
- **Multimodal**: CLIP, BLIP, LLaVA

Getting Started with Hugging Face

- Explore the Hub: huggingface.co
- Browse models, datasets, and Spaces.
- Install Libraries:

```
pip install transformers datasets tokenizers accelerate gradio
```

- Try a Pipeline:

```python
# Example: Sentiment Analysis
from transformers import pipeline
classifier = pipeline("sentiment-analysis")
result = classifier("Hugging Face is awesome!")
print(result)
# Example: Text Generation
generator = pipeline("text-generation")
output = generator("In a world of large language models,",
  max_length=50)
print(output)
```

Under the Hood

1. **Automatic model selection**
2. **Tokenization handled**
3. **Inference optimization**
4. **Result formatting**
5. **Device management**

Traditional Approach

1. Load tokenizer

2. Preprocess text

3. Load model

4. Run inference

5. Post-process results

... 50+ lines of code

Other Hugging face Pipelines

The Hugging Face `transformers` library supports a wide range of **pipelines**, each designed for a specific **natural language processing (NLP)** or **vision** task — so you can use powerful models without deep setup.

Pipeline Name	Task Description
"sentiment-analysis"	Classify sentiment (positive/negative)
"text-classification"	General text classification (multi-label or multi-class)
"zero-shot-classification"	Classify into labels **without training** on them
"text-generation"	Generate text (e.g., GPT models)
"text2text-generation"	Text-to-text tasks (e.g., summarization, translation)
"translation"	Translate between languages
"summarization"	Generate a summary of input text
"question-answering"	Extract answer from context
"fill-mask"	Predict missing word in a sentence (BERT-style)
"ner" (Named Entity Recognition)	Extract entities (like names, places, etc.)
"conversational"	Chatbot-style conversation
"sentence-similarity"	Measure similarity between two sentences
"token-classification"	Classify each token (used for NER, POS tagging, etc.)
"feature-extraction"	Extract embeddings/features from a model
"table-question-answering"	QA over structured data (tables)

➤ Sentiment Analysis

```python
pipeline("sentiment-analysis")("I love this!")
```

➤ Summarization

```python
pipeline("summarization")("Long article text goes here...")
```

➤ Translation

```python
pipeline("translation_en_to_fr")("This is amazing.")
```

➤ Question Answering

```python
qa = pipeline("question-answering")
qa({
    "question": "Where do pandas live?",
    "context": "Pandas are native to China and prefer bamboo forests."
})
```

To list all available pipelines in code:

```python
from transformers.pipelines import SUPPORTED_TASKS
print(SUPPORTED_TASKS.keys())
```

Open in Colab